Universiteit Gent - TELIN - DDCM
2024-2025

Workshop: Query-optimalisatie

1 Introductie

Tijdens de voorbije oefeningenlessen hebben jullie al heel wat geleerd over het gebruik van relationele (PostgreSQL) databanken. Daarnaast hebben jullie ook reeds een aantal lessen data-analyse, die hoofdzakelijk focusten op het schrijven van `SELECT`-queries in SQL (Structured Query Language), achter de rug. Waarschijnlijk hebben jullie al gemerkt dat je voor het oplossen van een bepaald probleem verschillende equivalente `SELECT`-queries kan opstellen die de gewenste data zullen opvragen. Ook al geven deze queries allemaal de gewenste data terug, vaak hanteren ze onderliggend een andere berekeningsmethode om tot die correcte oplossing te komen. Op kleine datasets heeft dit vaak heel weinig impact. Wanneer je echter grotere volumes data wil bevragen, kan de keuze voor een query een grote invloed hebben op de middelen die het databankbeheersysteem nodig heeft om deze query uit te kunnen voeren. Om jullie een idee te geven: het is mogelijk dat de uitvoering van een 'slechte' of 'niet-performante' query verschillende uren of gigabytes aan geheugen in beslag neemt, wat nefast is voor ondernemingen die grote hoeveelheden data verwerken. Je probeert dus best altijd een query te schrijven die zo performant mogelijk is in termen van uitvoeringstijd en geheugenvereisten. Daarom focussen we tijdens deze workshop op verschillende concepten om de performantie van een `SELECT`-query te meten en te analyseren. Daarnaast zullen we verschillende basistechnieken bestuderen die jullie kunnen helpen om voor een bepaald probleem een zo performant mogelijke query op te stellen. Deze technieken kunnen worden samengevat onder de term 'query-optimalisatie'.

2 Eerste stappen

Deze sectie dien je enkel door te nemen indien je nog niet (volledig) klaar bent met de vorige workshops of indien je wil vertrekken van onze modeloplossing. Indien je hier langer dan 15 minuten aan spendeert, vraag dan hulp aan een van de assistenten.

Voor deze workshop zullen we opnieuw werken met de `wielrennen` databank die in de vorige workshops reeds uitgebreid geïntroduceerd en gebruikt werd. Voor de volledigheid is het relationele databankschema van deze databank terug te vinden in de Appendix van deze workshop. Indien er nog geen correct geïmplementeerde of met data gevulde `wielrennen` databank op jullie lokale PostgreSQL cluster bestaat, kunnen jullie gebruik maken van een backup van deze databank in de vorm van een script met de naam `wielrennen_dml1.sql`. Dit script kunnen jullie terugvinden in de map van de workshop 'DML 1' op Ufora. Het inladen van deze backup (restore) op jullie lokale machine kan via de commandolijn-applicatie `psql` met het volgende commando (plaats het volledige commando op 1 lijn).

```
psql
    --host=127.0.0.1
    --port=5432
    --dbname=wielrennen
    --username=postgres
    --file=wielrennen_dml1.sql
```

Let wel op dat er reeds een lege `wielrennen` databank moet bestaan op je lokale PostgreSQL cluster. Deze dien je dus eerst zelf aan te maken.

Zorg, vooraleer verder te gaan, dat je een correct geïmplementeerde en met data gevulde `wielrennen` databank hebt aangemaakt op je lokale PostgreSQL cluster. Controleer zeker eens of elke tabel het verwachte aantal rijen bevat (zie workshop 'DML 1').

3 Kostanalyse

Vooraleer we verschillende technieken zullen introduceren om voor een bepaald probleem een zo performant mogelijke query op te stellen, is het belangrijk om te weten hoe je de performantie (of kost) van een `SELECT`-query kan meten en analyseren. Hiervoor zullen we onderstaande `SELECT`-query gebruiken als voorbeeld doorheen deze sectie.

```
SELECT * FROM wielrenner wr
    INNER JOIN wielerteam wt ON wr.teamnaam = wt.naam
    INNER JOIN uitslag u ON wr.naam = u.rennernaam;
```

Het is eenvoudig om te zien dat deze query alle data uit de tabellen `wielrenner`, `wielerteam` en `uitslag` combineert door gebruik te maken van de vreemde sleutelreferenties die gedefinieerd zijn tussen deze tabellen. Als resultaat zal deze query voor elke wielrenner alle data met betrekking tot (i) deze wielrenner, (ii) het team waartoe deze wielrenner behoort en (iii) alle uitslagen van deze wielrenner teruggeven. In eerste instantie zouden we zelf eens kunnen redeneren over hoe deze query zal worden uitgevoerd. Eerst en vooral wordt er een `INNER JOIN`-operatie uitgevoerd om de rijen uit de tabellen `wielrenner` (934 rijen) en `wielerteam` (32 rijen) te koppelen op basis van gelijke waarden voor de attributen die corresponderen met de naam van het wielerteam. Daarvoor wordt elke rij in de tabel `wielrenner` overlopen en wordt er een match gezocht (op basis van een gelijk teamnaam) met een rij in de tabel `wielerteam`. Indien er een match wordt gevonden, wordt de gecombineerde rij teruggegeven. Na het overlopen van elke rij in de tabel `wielrenner` is er dus een tussentijds resultaat in de vorm van een tabel opgebouwd. Hierna wordt dit tussentijds resultaat gekoppeld met de rijen uit de tabel `uitslag` (14731 rijen) op basis van gelijke waarden voor de attributen die corresponderen met de naam van de wielrenner. Opnieuw wordt dezelfde procedure gevolgd en na het uitvoeren van procedure zal het eindresultaat van deze query worden teruggegeven.

> Voer bovenstaande query uit en analyseer het tussentijds resultaat en het eindresultaat. Klopt dit met je verwachtingen? Is het mogelijk om de volgorde van de `JOIN`-operaties te wijzigen zonder dat het eindresultaat wijzigt?

3.1 `EXPLAIN`

Naast het zelf redeneren over de uitvoering van een query, kan je ook echt gaan opvragen op welke manier PostgreSQL een query intern zal uitvoeren. Dit kan je doen door het `EXPLAIN`[1]-keyword voor een query te plaatsen. In dit geval krijg je geen resultatentabel te zien bij uitvoering van de query, omdat PostgreSQL de query niet echt zal uitvoeren. Wel krijg je het *uitvoerings-* of *queryplan* te zien dat onderliggend door PostgreSQL gebruikt zal worden bij de verwerking van de query.

[1] `https://www.postgresql.org/docs/current/using-explain.html`

Voer bovenstaande SELECT-query, voorafgegaan door het EXPLAIN-keyword, uit. Bestudeer kort het verkregen queryplan.

Hieronder wordt het queryplan van deze query weergegeven[2].

```
QUERY PLAN
Hash Join  (cost=34.73..484.68 rows=14731 width=244)
    Hash Cond: ((wr.teamnaam)::text = (wt.naam)::text)
    ->  Hash Join  (cost=33.02..439.17 rows=14731 width=152)
        Hash Cond: ((u.rennernaam)::text = (wr.naam)::text)
        ->  Seq Scan on uitslag u  (cost=0.00..367.31 rows=14731 width=85)
        ->  Hash  (cost=21.34..21.34 rows=934 width=67)
            ->  Seq Scan on wielrenner wr  (cost=0.00..21.34 rows=934 width=67)
    ->  Hash  (cost=1.32..1.32 rows=32 width=92)
        ->  Seq Scan on wielerteam wt  (cost=0.00..1.32 rows=32 width=92)
```

Om het plan correct te interpreteren moet je dit plan lezen van binnen (het diepste niveau) naar buiten (of, met andere woorden, van het meest rechtse niveau naar het meest linkse niveau). In dit voorbeeld wordt er eerst een hash-functie uitgevoerd op elke rij van de tabel `wielrenner`. Hierna wordt de tabel `uitslag` sequentieel overlopen en wordt er, voor elke rij in deze tabel, een match gezocht in de tabel `wielrenner` door middel van de gebruikte hash-functie en de gedefinieerde `Hash Conditie`. Dit resulteert in een `Hash Join` die uitgevoerd wordt op de tabellen `wielrenner` en `uitslag`. Het volstaat om te weten dat een `Hash Join` een algoritme is om efficiënt een `JOIN`-operatie uit te voeren. Daarna wordt er opnieuw een `Hash Join` uitgevoerd op het tussenresultaat en de tabel `wielerteam`.
Je merkt dus dat de uitvoeringsvolgorde die PostgreSQL zal hanteren (eerst het combineren van rijen uit de tabellen `wielrenner` en `uitslag`, en dan pas uit de tabel `wielerteam`) niet overeenkomt met de uitvoeringsvolgorde die wij hadden vooropgesteld, en dus oorspronkelijk hadden verwacht. De reden hiervoor is dat PostgreSQL verschillende plannen om een query uit te voeren overweegt, en het plan kiest dat resulteert in de uitvoering met de laagste kost (484.68). Hieronder zullen wij dieper ingaan op de betekenis en de berekening van deze kost.

3.2 Query planner

Naast alle uit te voeren operaties wordt er in het uitvoeringsplan ook, per operatie, een lijst van metrieken gegeven. Omdat een query, voorafgegaan door het EXPLAIN-

[2] Het is in pgAdmin 4 ook mogelijk om een visueel overzicht te krijgen van dit queryplan door op de 'Explain'-knop te drukken bovenaan in de query tool.

keyword, niet echt wordt uitgevoerd, zijn de waarden die deze metrieken aannemen schattingen die worden berekend op basis van statistieken (bv. het aantal rijen) die PostgreSQL bijhoudt over de databank. Deze statistieken worden bij iedere operatie (bv. toevoegen van data, selecteren van data. . .) die effectief uitgevoerd wordt op de databank geüpdatet. Neem, bijvoorbeeld, de metrieken die we zien voor de laatste `Hash Join` in het queryplan.

- **cost=34.73..484.68**: De geschatte kost voor de uitvoering van de operatie (uitgedrukt in een arbitraire eenheid).
- **rows=14731**: Het geschatte aantal rijen dat deze operatie zal teruggeven.
- **width=244**: Het geschatte aantal bytes waaruit elke rij, die door de operatie zal worden teruggeven, zal bestaan.

Zoals je ziet wordt de kost voorgesteld door twee waarden. De eerste waarde (34.73) is de *start-up kost* en geeft aan wat de kost is om de *eerste rij* terug te geven wanneer de operatie wordt uitgevoerd. De tweede waarde (484.68) is de *totale kost* die nodig is om *alle rijen* terug te geven na uitvoering van de operatie. De precieze berekening van de kost is vrij complex en valt buiten de scope van dit vak. Het is wel belangrijk om te onthouden dat zowel het *geheugenverbruik* alsook de *totale uitvoeringstijd* van de operatie een belangrijke rol spelen in deze berekening. Met de inzichten die we hierboven hebben verworven kunnen we besluiten dat het optimaliseren van queries niet enkel de taak is van de gebruiker, maar ook van het databankbeheersysteem zelf. Intern zal PostgreSQL telkens een zogenaamde *query planner/optimizer* gebruiken om een query zo performant mogelijk uit te voeren. De reden dat PostgreSQL een dergelijke query planner gebruikt, toont aan dat voor heel wat applicaties het geheugenverbruik en de totale uitvoeringstijd van een query vrij cruciaal zijn. Op basis van de opgegeven query en een groot aantal gekende optimalisatietechnieken genereert de query planner dus verschillende uitvoeringsplannen. Daarna kiest de planner het plan met de laagste geschatte kost om te gebruiken tijdens de uitvoering. Veel optimalisaties die je op het eerste zicht zou kunnen doorvoeren hebben dus mogelijks helemaal geen effect op de kost, omdat het databankbeheersysteem deze optimalisaties intern zelf al zal doorvoeren.

3.3 `EXPLAIN ANALYZE`

Zoals reeds vermeld werd in Sectie 3.1 en 3.2 houdt de query planner enkel rekening met schattingen op basis van statistieken die je kan opvragen door middel van het `EXPLAIN`-keyword. Deze schattingen geven soms een vertekend beeld over het geheugenverbruik en de uitvoeringstijd van een query. Je kan echter, voor elke operatie die intern uitgevoerd zal worden, ook een aantal exacte waarden opvragen.

Hiervoor dien je de query uit te voeren voorafgegaan door het `EXPLAIN ANALYZE`-keyword. In tegenstelling tot bij het gebruik van `EXPLAIN`, wordt een query, voorafgegaan door het `EXPLAIN ANALYZE`-keyword, wel degelijk uitgevoerd.

> Voer bovenstaande `SELECT`-query, voorafgegaan door het `EXPLAIN ANALYZE`-keyword, uit. Bestudeer kort het verkregen queryplan.

Als je de query, voorafgegaan door het `EXPLAIN ANALYZE`-keyword, uitvoert, zie je op het eerste zicht min of meer hetzelfde queryplan als voorheen. Er worden echter, samen met de geschatte waarden, ook een aantal exacte waarden voor bepaalde metrieken weergegeven. De metrieken waarvoor dit gebeurt zijn de volgende.

- **actual time**: De tijd die nodig was voor de uitvoering van de operatie (uitgedrukt in ms). Er wordt opnieuw een onderscheid gemaakt tussen start-up tijd en totale tijd.
- **rows**: Het exacte aantal rijen dat door de operatie werd teruggeven.
- **loops**: Het aantal keer dat de operatie werd uitgevoerd.
- **Planning time**: De tijd die de query planner nodig had om het meest optimale queryplan te selecteren (uitgedrukt in ms).
- **Execution time**: De totale tijd die nodig was om de query uit te voeren en het resultaat terug te geven (uitgedrukt in ms).

Vaak is het beter om het `EXPLAIN ANALYZE`-keyword te gebruiken, omwille van de volgende redenen. Ten eerste krijg je exacte informatie in verband met de uitgevoerde query in plaats van schattingen. Daarnaast worden de statistieken, die gebruikt worden voor het maken van schattingen door de query planner, geüpdatet, omdat de query effectief wordt uitgevoerd. Het gebruik van `EXPLAIN ANALYZE` zal, met andere woorden, typisch ook de accuraatheid van de schattingen verbeteren, indien deze nog niet accuraat genoeg waren. Enkel wanneer een query heel lang duurt of bijzonder zwaar is, kan je best terugvallen op het gebruik van het `EXPLAIN`-keyword.

4 Optimalisatietechnieken

Naast de optimalisaties die toegepast worden door de query planner van PostgreSQL bestaan er ook extra technieken die je zelf kan toepassen om een query te optimaliseren. Je zal merken dat er geen eenduidig wondermiddel bestaat dat

voor iedere query zal werken. Of en wanneer je bepaalde technieken kan inzetten hangt heel hard af van de context en vereist enige ervaring. Onthoud dit dus goed wanneer je probeert te begrijpen waarom een bepaalde techniek wel een invloed heeft op de performantie van één query en geen invloed heeft op de performantie van een andere query.

4.1 Indexering

Beschouw de volgende `SELECT`-query.

```
SELECT * FROM uitslag WHERE positie = 1;
```

> Vraag het queryplan van bovenstaande query op door middel van het `EXPLAIN ANALYZE`-keyword. Onthoud de kost van de volledige query.

Zoals je kan zien voert PostgreSQL een sequentiële scan uit waarmee het voor iedere rij in de `uitslag` tabel controleert of de bijhorende positie gelijk is aan 1. De rijen waarvoor dit niet het geval is worden weggefilterd. Zoals je kan zien zijn dit er 14631 om uiteindelijk slechts 100 rijen over te houden. Ondanks dit kleine aantal rijen moet de waarde van het `positie` attribuut voor alle 14731 rijen in de tabel `uitslag` gecontroleerd worden. Het is duidelijk dat dit efficiënter moet kunnen.
Een typische oplossing voor een dergelijk probleem is het definiëren van een *index*. Bij de definitie van een index wordt er een nieuwe datastructuur (een opzoektabel) aangemaakt die het mogelijk maakt om, voor de attributen waarop de index gedefinieerd is, heel snel rijen met bepaalde waarden voor deze attributen terug te vinden. Je kan het vergelijken met een opzoektabel (vaak ook index genoemd) zoals die typisch voorkomt in kookboeken. In zo'n index worden, per ingrediënt, de paginanummers opgelijst van de pagina's waarop recepten te vinden zijn waarin dat ingrediënt wordt gebruikt. Indien deze index niet beschikbaar zou zijn en je te weten zou willen komen in welke recepten er 'tomaten' gebruikt worden, dan zou je het hele boek moeten doorbladeren om alle recepten te controleren (wat overeenkomt met een sequentiële scan). Dankzij de index kan je echter rechtstreeks naar de juiste recepten springen. Dit is exact wat een index in een databank ook doet. Per waarde (of combinatie van waarden) voor een bepaalde attribuut (of voor een combinatie van attributen) waarop de index gedefinieerd is, wordt er gerefereerd naar de rijen (locaties in het RAM-geheugen) die deze waarde(n) als data hebben opgeslagen.
Om een index met naam `indexnaam` aan te maken op de combinatie van attributen $\texttt{attr}_1,\ldots,\texttt{attr}_n$ van de tabel met naam `basisrelatie`, kan je onderstaande

instructie[3] uitvoeren.

```
CREATE INDEX indexnaam ON basisrelatie (attr_1,...,attr_n);
```

Om een index met naam `indexnaam` te verwijderen, kan je dan weer eenvoudigweg onderstaande instructie[4] uitvoeren.

```
DROP INDEX indexnaam;
```

Maak een index aan op het attribuut `positie` van de tabel `uitslag`. Bekijk vervolgens opnieuw het queryplan van de hierboven gegeven query met behulp van het `EXPLAIN ANALYZE`-keyword. Vergelijk de kost van de volledige query met het vorige resultaat.

Je merkt dat de kost van de query (veel) lager is na het definiëren van deze index. Dit komt omdat PostgreSQL nu beslist om in plaats van een sequentiële scan, een (bitmap) index scan uit te voeren op de gedefinieerde index. Er wordt, met andere woorden, gebruik gemaakt van deze index om de correcte data terug te vinden.
Een index kan dus de performantie van een `SELECT`-query enorm verbeteren. Daarom lijkt het misschien een goed idee om op alle mogelijke combinaties van attributen een index te definiëren, om op elke mogelijke manier waarop de databank bevraagd kan worden te anticiperen. Dit is echter geen goed idee, aangezien het definiëren van een index ook nadelen met zich mee kan brengen. Het meest voor de hand liggende nadeel is dat voor iedere index die je definieert, een nieuwe datastructuur wordt aangemaakt waarin, voor iedere waarde-combinatie die voorkomt, verwijzingen naar de corresponderende rijen opgeslagen zitten. Aangezien zo'n datastructuur zelf ook moet worden opgeslagen in het geheugen, kan dit de geheugenvereisten van een (grote) databank enorm verhogen. Een tweede nadeel is dat het aantal mogelijke indices exponentieel stijgt met het aantal attributen in een tabel, aangezien je ook indices over combinaties van attributen kan definiëren. Daarnaast kan je ook indices aanmaken op bepaalde waardetransformaties, op een beperkt aantal waarden van een attribuut... Je merkt dus dat het aantal mogelijkheden om indices te definiëren eindeloos zijn. Tot slot is het ook belangrijk om te vermelden dat een index redelijk wat onderhoudswerk vergt. Elke keer dat je een nieuwe rij met een attribuut waarop een index gedefinieerd is toevoegt (`INSERT`), of een attribuutwaarde van een bestaande rij waarop een index gedefinieerd is aanpast (`UPDATE`), dient de index ook aangepast te worden. We kunnen dus concluderen dat indices de

[3] `https://www.postgresql.org/docs/current/sql-createindex.html`
[4] `https://www.postgresql.org/docs/current/sql-dropindex.html`

performantie van bepaalde (zware) operaties kunnen versnellen, maar ook de performantie van andere operaties significant kunnen vertragen. Denk daarom steeds goed na of het nuttig is om een index te definiëren (bv. wanneer een zware `SELECT`-query zeer vaak uitgevoerd zal worden in een databank, en er over het algemeen weinig `INSERT`-operaties gebeuren...).
Tot slot willen we nog een aantal tips meegeven die zouden kunnen helpen bij het beslissen over het al dan niet definiëren van een index.

- Het kan nuttig zijn om op een attribuut dat vaak gebruikt wordt in booleaanse condities in een `WHERE`- of `JOIN`-clausule een index te definiëren. Zoals hierboven reeds is aangetoond kan dit de performantie van een `SELECT`-query namelijk sterk verbeteren. In het bijzonder zijn (combinaties van) attributen waarop `UNIQUE`-beperkingen gedefinieerd zijn (bv. primaire sleutels) belangrijk in dit opzicht. Deze attributen worden typisch vaak gebruikt in filters (omdat de onderliggende waarden uniek zijn, resulteert dit dus ook in een grote reductie van het aantal rijen) en in `JOIN`-condities om tabellen te koppelen op basis van vreemde sleutel-referenties. Merk op dat, omwille van deze reden, PostgreSQL bij het aanmaken van een `UNIQUE`-beperking automatisch een index op de combinatie van attributen die meespelen in deze beperking definieert.

- Als een booleaanse propositie bestaat uit conjuncties (sleutelwoord `AND`) van condities op attributen binnen dezelfde tabel (bv. `positie = ... AND tijd = ...`), definieer dan een index op de combinatie van deze attributen (bv. op `positie` en `tijd` samen). De reden hiervoor is dat PostgreSQL dan efficiënt op zoek kan gaan naar rijen waarin alle opgegeven waarden voorkomen.

- Wanneer een attribuut zeer veel verschillende waarden bevat, kan je verwachten dat er een laag aantal rijen zal worden teruggegeven als resultaat van een `SELECT`-query die rijen behoudt met een specifieke waarde voor dit attribuut. Wanneer je weinig rijen verwacht in het eindresultaat kan een index zeer nuttig zijn. Opnieuw zijn combinaties van attributen waarop `UNIQUE`-beperkingen gedefinieerd zijn belangrijk in dit opzicht. Als je een filter gebruikt (bv. in de `WHERE`-clausule) op zo'n combinatie, moet het databanksysteem slechts 1 geheugenblok vanuit het RAM-geheugen inladen waarin de gevraagde rij wordt opgeslagen. Indien je geen index gebruikt moet het databanksysteem in het slechtste geval elk geheugenblok vanuit het RAM-geheugen inladen om te kijken of de gevraagde rij zich daarin bevindt. Stel, daarnaast, dat je een index definieert op een attribuut met als datatype `boolean`. In dit geval kan je verwachten dat gemiddeld gezien 50% van de rijen de waarde `true` zal bevatten en 50% van de rijen de waarde `false` zal bevatten. Wanneer je filtert op basis van een waarde voor dit attribuut, moet het databanksysteem nog steeds gemiddeld 50% van de rijen inladen, die verspreid opgeslagen kunnen zijn in het RAM-geheugen, wat de performantiewinst dus beperkter maakt.

- Het definiëren van een index op attributen
- in tabellen die weinig data bevatten, heeft weinig nut. Zulke tabellen passen namelijk in één enkel geheugenblok, dat in een keer kan worden ingeladen.
- Zoals we reeds hebben aangehaald, moet je steeds de meest voorkomende operaties binnen een databank beschouwen, vooraleer je beslist over welke indices er een positieve impact kunnen hebben.

4.2 Berekeningen en transformaties in booleaanse condities

In Sectie 4.1 hebben we aangetoond dat de kost van een query significant verlaagd kan worden door gebruik te maken van een index, zeker wanneer er vaak rijen opgevraagd moeten worden die voldoen aan bepaalde booleaanse condities in de `WHERE`- of `JOIN`-clausule. Vaak is het echter zo dat er geen exacte attribuutwaarden worden gebruikt in deze condities, maar er berekeningen worden gedaan met of transformaties worden toegepast op de betrokken attributen. Een voorbeeld hiervan is onderstaande query, waarmee er gezocht wordt naar alle wielrenners die deel uitmaken van een team waarvan de naam eindigt met de deelstring 'Team'.

```
SELECT * FROM wielrenner
    WHERE substring(teamnaam, position('Team' in teamnaam)) = 'Team';
```

> Vraag het queryplan van bovenstaande query op door middel van het `EXPLAIN ANALYZE`-keyword. Onthoud de kost van de volledige query.

Opnieuw beslist PostgreSQL om sequentieel alle rijen in de tabel `wielrenner` te overlopen, zelfs nadat je een index gedefinieerd hebt op het attribuut `teamnaam` van de tabel `wielrenner` (controleer dit). De reden hiervoor is dat het definiëren van een index op de exacte waarden van dit attribuut niet helpt wanneer een query rijen filtert op basis van transformaties (in dit geval een substring). Gelukkig is het ook mogelijk om een index aan te maken op berekeningen met of transformaties op attributen (bv. wiskundige berekeningen, stringfuncties, wiskundige functies, datum/tijd functies. . .).

> Maak een index aan op de transformatie `substring(teamnaam, position('Team' in teamnaam))` in de tabel `wielrenner`. Vraag vervolgens opnieuw het queryplan van de hierboven gegeven query op door middel van `EXPLAIN ANALYZE`-keyword. Vergelijk de kost van de volledige query met het vorige resultaat.

Je merkt dat, na het definiëren van deze index, de sequentiële scan over alle wielrenners niet meer nodig is, maar de query planner van PostgreSQL opnieuw kiest voor een (bitmap) index scan, waardoor de kost dus verlaagt. Merk hierbij wel op dat, wanneer er zeer frequent queries met berekeningen of transformaties uitgevoerd worden, je er tijdens het databankontwerp misschien beter voor had gekozen om de berekeningen of transformaties als afgeleide data op te slaan.

Verwijder alle indices die je in de vorige oefeningen hebt gedefinieerd.

4.3 Vermijd overbodige attributen en joins

De kost (of, beter gezegd, het geheugenverbruik) van een `SELECT`-query kan je relatief eenvoudig verlagen door het aantal (overbodige) attributen in de resultatentabel te limiteren. Zeker wanneer je een groot aantal tabellen combineert door middel van `JOIN`-operaties, kan de uiteindelijke resultatentabel bestaan uit (zeer) veel attributen. In het bijzonder kan het gebruik van het `USING`-keyword in het geval van zware `JOIN`-operaties een positieve impact hebben op het geheugenverbruik (waarom is dit?). Daarnaast kan je je best limiteren tot enkel de attributen waarvan je effectief data wil opvragen. Herinner je ook de 'width' metriek in het queryplan, die het geschatte aantal bytes waaruit elke rij, die door de query zal worden teruggeven, zal bestaan weergeeft.

Voer onderstaande queries uit en vraag de queryplannen van deze queries op door middel van het `EXPLAIN ANALYZE`-keyword. Merk je een verschil op in resultaat, kost, uitvoeringstijd of geheugenvereisten van deze queries? Hoe komt dit?

```
SELECT * FROM uitslag u1
    INNER JOIN uitslag u2 ON u1.wedstrijdnaam = u2.wedstrijdnaam
        AND u1.ritnr = u2.ritnr
WHERE u1.rennernaam != u2.rennernaam;

SELECT u1.rennernaam, u1.positie, u2.rennernaam, u2.positie FROM uitslag u1
    INNER JOIN uitslag u2 USING (wedstrijdnaam, ritnr)
WHERE u1.rennernaam != u2.rennernaam;
```

Wanneer je de queryplannen van beide queries aandachtig bekijkt, zal je zien dat de kost van deze queries niet verschilt, maar de uitvoering van de tweede query iets geheugenefficiënter is. De reden hiervoor is dat, door het gebruik van het `USING`-keyword in het geval van de tweede query, equivalente attributen (bv. `wedstrijdnaam` en `ritnr`) worden gecombineerd tot een enkel attribuut in de resultatentabel.
Naast het beperken van het aantal opgevraagde attributen kan je ook `JOIN`-operaties die overbodig zijn best weglaten.

> Voer onderstaande queries uit en vraag de queryplannen van deze queries op door middel van het `EXPLAIN ANALYZE`-keyword. Merk je een verschil op in resultaat, kost, uitvoeringstijd of geheugenverbruik van deze queries? Hoe komt dit?

```
SELECT
    wedstrijdnaam,
    ritnr,
    positie,
    rennernaam,
    teamnaam
FROM uitslag u
    INNER JOIN wielrenner wr ON u.rennernaam = wr.naam
    INNER JOIN wielerteam wt ON wr.teamnaam = wt.naam;

SELECT
    wedstrijdnaam,
    ritnr,
    positie,
    rennernaam,
    teamnaam
FROM uitslag u
    INNER JOIN wielrenner wr ON u.rennernaam = wr.naam;
```

Als je enkel de data van de attributen `wedstrijdnaam`, `ritnr`, `positie`, `rennernaam` en `teamnaam` wil opvragen, is het helemaal niet nodig om nog eens een `JOIN`-operatie uit te voeren met de tabel `wielerteam`. Denk dus steeds goed na of je echt alle opgevraagde attributen en `JOIN`-operaties nodig hebt. Let wel op dat je na het weglaten van geselecteerde attributen en `JOIN`-operaties nog steeds de gewenste data bekomt.

4.4 Gebruik van `DISTINCT` en `ORDER BY`

Tijdens de SQL-lessen heb je geleerd dat het toevoegen van het `DISTINCT`-keyword in de `SELECT`-clausule ervoor zorgt dat alle rijen in de resultatentabel slechts een keer worden getoond.

> Vraag, door middel van het `EXPLAIN ANALYZE`-keyword, het queryplan op dat PostgreSQL genereert voor de query waarmee je alle unieke namen van wielrenners die ooit in de top 10 van een rit zijn gefinisht opvraagt.

Wanneer je het queryplan van deze query bekijkt, zou je moeten vaststellen dat er eerst een sequentiële scan wordt uitgevoerd op de tabel `uitslag`. Tijdens de `HashAggregate` stap die hierna volgt worden alle rijen verdeeld in groepen met gelijke waarden voor het attribuut `rennernaam`, en wordt er per groep 1 rij teruggegeven. Zelfs al kan je de sequentiële scan vermijden door het definiëren van een index op attribuut `positie`, dan nog is het vergelijken van de verschillende waarden door middel van de `HashAggregate` stap een heel dure operatie. Denk daarom altijd na of het mogelijk is om het gebruik van het `DISTINCT`-keyword te vermijden door attributen te selecteren (soms aanvullend op de attributen waarvan je in eerste instantie data wil opvragen), die zeker resulteren in unieke rijen (bv. de primaire sleutel).

> Sorteer de resultaten van de vorige query in alfabetisch oplopende volgorde. Vergelijk de kost van deze query met het vorige resultaat.

Net als het dedupliceren van rijen in de resultatentabel, is het sorteren van rijen op basis van waarden van een of meerdere attributen een zeer dure operatie, omwille van de rekenintensieve sorteeralgoritmen (bv. quicksort). Indien het dus niet noodzakelijk is om resultaten gesorteerd terug te geven, is het beter om het gebruik van `ORDER BY` te vermijden. In tegenstelling tot het `DISTINCT`-keyword is er voor de `ORDER BY`-clausule bijna nooit een performant alternatief.

4.5 En verder...

Naast de technieken die we hierboven hebben aangehaald bestaan er nog een heleboel andere technieken met betrekking tot meer geavanceerde SQL-concepten. Zo zal het, bijvoorbeeld, vaak helpen om gecorreleerde subqueries (SQL, reeks 3) te vervangen door alternatieve, niet-gecorreleerde subqueries of door `JOIN`-operaties (indien mogelijk). Ook het gebruik van aggregatiefuncties (bv. `= MAX`) in plaats van

subquery-operatoren (bv. `>= ALL`) voor het vergelijken van rijen met een geaggregeerde waarde (SQL, reeks 4) zal normaal gezien resulteren in performantiewinst. We willen echter nogmaals benadrukken dat er helaas geen wondermiddel bestaat dat voor iedere query zal werken, en dat dit dus zeer sterk afhangt van de databankstructuur, de data, de query, de geheugenopslag. . .
Tot slot willen we nog kort vermelden dat er ook tijdens de ontwerpsfase van de databank reeds kan worden ingespeeld op de performantie van queries. In Sectie 4.2 gaven wij al aan dat het soms performanter is om berekeningen of transformaties als afgeleide data op te slaan. Ook kan je ervoor kiezen om twee tabellen fysiek ongescheiden op te slaan (dus samen te voegen tot één tabel) om dure, veel voorkomende `JOIN`-operaties te vermijden. Dit proces heet denormalisatie. Let op, beide ingrepen zullen leiden tot de opslag van redundante data, wat op zijn beurt dan weer kan leiden tot inconsistenties. Doe dit dus steeds uiterst bedachtzaam. Je zou, tenslotte, ook kunnen overwegen om een relationele databank in te ruilen voor een NoSQL databank om performantiewinst te boeken. . .

5 Oefeningen

Als afsluiter van deze workshop, vragen we jullie om een aantal eenvoudige oefeningen op te lossen waarvoor jullie een `SELECT`-query moeten schrijven. De bedoeling is nu niet alleen om deze oefeningen correct op te lossen, maar ook om zo performant mogelijke queries te schrijven. Om jullie een idee te geven van hoe 'performant' jouw oplossing is, hebben we bij iedere query een richtkost opgelijst (die, afhankelijk van de laptop waarop de query wordt uitgevoerd, licht kan verschillen). Kunnen jullie een query opstellen met een lagere kost dan deze richtkost?

1. Geef alle uitslagen waarbij het verschil tussen de tijd en de bonustijd strikt meer dan 4 uur bedraagt. Lijst, voor elk van deze uitslagen, de wedstrijdnaam, het ritnummer en de rennernaam op in het eindresultaat.

   Richtkost: 367.99

2. Geef de naam en de geboortedatum van alle wielrenners die geboren zijn in de maand februari van het jaar 1994.

   Richtkost: 8.29

3. Tot slot hebben wij van de organisatie die instaat voor het beheer van de `wielrennen` databank een `SELECT`-query ontvangen waarmee de data die beantwoorden aan onderstaande vereisten opgevraagd kunnen worden. Kunnen jullie hen als databankexpert helpen om deze query te optimaliseren. Let op, interpreteer de gegeven query eerst en controleer of deze query een correct resultaat teruggeeft, want dit is geen garantie.

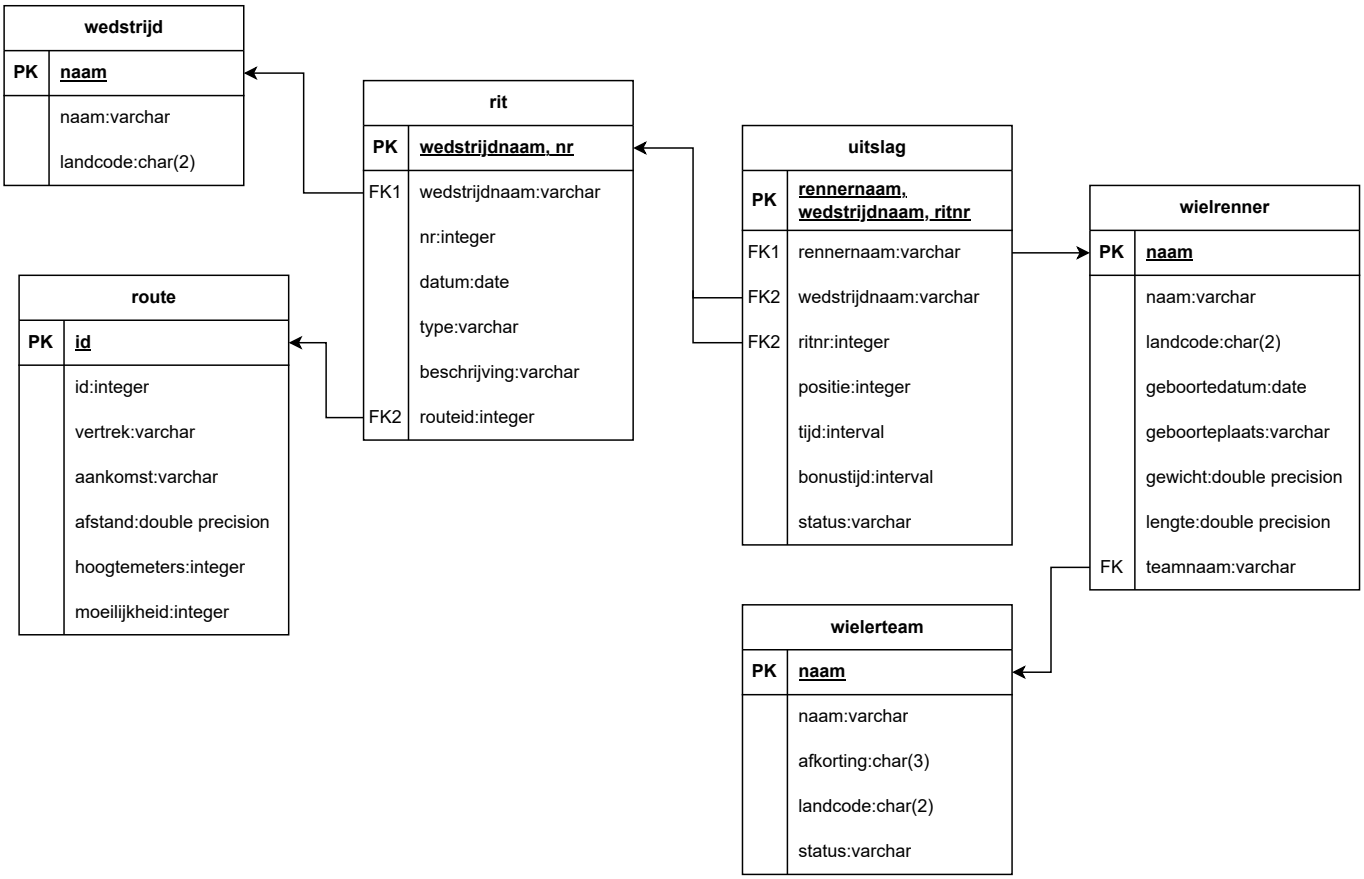
Lijst, per rit, alle namen van teams waarvan er minstens 3 renners in de top 5 van die rit zijn gefinisht op. Enkel ritten waarvoor er zo'n team bestaat moeten opgenomen worden in het eindresultaat. Daarnaast worden er in het eindresultaat unieke rijen en drie attributen verwacht: `wedstrijdnaam`, `ritnr` en `teamnaam`.

```
SELECT DISTINCT u1.wedstrijdnaam, u1.ritnr, wt1.naam as teamnaam
FROM uitslag u1
    INNER JOIN wielrenner wr1 ON u1.rennernaam = wr1.naam
    INNER JOIN wielerteam wt1 ON wr1.teamnaam = wt1.naam
    INNER JOIN uitslag u2 ON u1.wedstrijdnaam = u2.wedstrijdnaam
        AND u1.ritnr = u2.ritnr
        AND u1.rennernaam != u2.rennernaam
    INNER JOIN wielrenner wr2 ON u2.rennernaam = wr2.naam
    INNER JOIN wielerteam wt2 ON wr2.teamnaam = wt2.naam
        AND wt1.naam = wt2.naam
    INNER JOIN uitslag u3 ON u1.wedstrijdnaam = u3.wedstrijdnaam
        AND u1.ritnr = u3.ritnr
        AND u1.rennernaam != u3.rennernaam
        AND u2.rennernaam != u3.rennernaam
    INNER JOIN wielrenner wr3 ON u3.rennernaam = wr3.naam
    INNER JOIN wielerteam wt3 ON wr3.teamnaam = wt3.naam
        AND wt1.naam = wt3.naam
WHERE u1.positie <= 5
    AND u2.positie <= 5
    AND u3.positie <= 5;
```

Richtkost: 282.68

Appendix: Relationeel schema `wielrennen` databank

In onderstaande figuur vind je het relationeel databankschema van de `wielrennen` databank. Hierin wordt iedere basisrelatie weergegeven door een rechthoek, die bovendien een oplijsting van alle attributen met bijhorende datatypes bevat. Daarnaast worden de attributen die behoren tot de primaire sleutel (PK) bovenaan weergegeven, en worden vreemde sleutels (FKx) voorgesteld door een pijl tussen de betreffende attribuutverzamelingen. Alle extra beperkingen die niet kunnen worden weergegeven in dit schema, worden hieronder opgelijst.



Extra beperkingen

- `wielerteam`:
  - uniek: {afkorting}
  - check: status ∈ {‘World Tour’, ‘Pro Tour’}
- `wielrenner`:
  - optioneel: geboorteplaats, gewicht, lengte
  - check: gewicht > 0
  - check: lengte > 0
- `route`:
  - check: afstand > 0
  - check: hoogtemeters ≥ 0
  - check: moeilijkheid ∈ [1,5]
- `rit`:
  - check: nr ≥ 1
  - check: type ∈ {‘Road Race’, ‘Individual Time Trial’, ‘Team Time Trial’}
- `uitslag`:
  - optioneel: positie, tijd, bonustijd
  - check: positie > 0
  - check: tijd > 0
  - check: bonustijd ≥ 0
  - check: status ∈ {‘Did Finish’, ‘Did Not Finish’,‘Did Not Start’, ‘Over Time Limit’, ‘Disqualified’}
  - check: status = ‘Did Finish’ ⇒ positie is not null ∧ tijd is not null ∧ bonustijd is not null
  - check: status ≠ ‘Did Finish’ ⇒ positie is null
  - check: status ∈ {‘Did Not Finish’, ‘Did Not Start’} ⇒ tijd is null
  - check: status = ‘Did Not Start’ ⇒ bonustijd is null
  - trigger: een wielrenner kan op eenzelfde datum slechts aan 1 rit deelnemen
  - trigger: indien de uitslagstatus van een renner in een rit niet gelijk is aan ‘Did Finish’, kan hij/zij niet meer deelnemen aan ritten in dezelfde wedstrijd met een hoger volgnummer